Mackenzie Tjogas
402

HW Assignment #3:

1.) Stephens 7.1
The original comments in the code explain accurately what the code does but don't explain how or why, they simply state the obvious happenings of the code. You could git rid of most of the comments and just add a link to the wikipedia page:

```
// Use Euclid's algorithm to calculate the GCD.
// See en.wikipedia.org/wiki/Euclidean_algorithm for more info if needed.
private long GCD( long a, long b )
{
  a = Math.abs( a );
  b = Math.abs( b );

  for( ; ; )
  {
    long remainder = a % b;
    If( remainder == 0 ) return b;
    a = b;
    b = remainder;
  };
}
```

2.) Stephens 7.2
These comments could have been generated if the programmer could have been following a top-down approach which involves describing the code, even when it gets redundant to describe what each code statement does. The programmer also could have created these comments because he/she wrote them after having written the code. This strategy can often result in a very surface-level description of the what and not including the how of the code.

3.) Stephens 7.4
The code written for exercise 7.3 already validates the inputs and the results. There is also a **Debug.Assert** method that throws an exception if a problem arises. I would say that this code already is offensive.

4.) Stephens 7.5
Error handling code could be added but it may not be the right decision to add it. The calling code should handle errors that arise and this is how the current implementation handles errors, so additional error handling code isn't needed.

5.) Stephens 7.7

a. Walk to car
b. Get into the car
c. Start the car
d. Back out of the driveway
e. Drive straight down the block
f. Make a right turn
g. Make the next left turn
h. Make another left at the stoplight
i. Drive straight until the next major stoplight. Then turn right
j. Turn left into the parking lot
k. Find a parking spot and park in it
l. Stop the car and exit the car
m. Lock the car and walk into the supermarket

Assumptions:

-The car is parked facing the garage (and therefore needs to be backed out)
-The car doesn't need gas
-My car is not blocked or parked in by another car in the driveway
-The parking lot has empty spots

6.) Stephens 8.1

```
//To check my answers here is a second method that is less efficient
SeeIfTheyArePrime method(int a, int b)
        // Returns a boolean declaring if a and b are relatively prime
        // use positive numbers
        a = absoluteValue(a)
        b = absoluteValue(b)
        if a or b is 1:
                return True
        if a or b is 0:
                return False
        //figure out which is smaller, a or b
        smallerVal = minimum(a, b)
        for (int i = 2; i <= smallerVal; i++) {
                if (the remainder of a and i is 0) && (the remainder of b and i is 0) {
                        return False
                }
                return True
        }
```

//Now for the **IsRelativelyPrime** method
The tests that the method would perform would be along the lines of:

For 1,000 tries pick random a and b and then:
    Assert SeeIfTheyArePrime(a, b) =
        IsRelativelyPrime(a, b)

For 1,000 tries pick random a and then:
    Assert SeeIfTheyArePrime(a, a) =
        IsRelativelyPrime(a, a)

For 1,000 tries pick random a and then:
    Assert SeeIfTheyArePrime(a, 1) relatively prime
    Assert SeeIfTheyArePrime(a, -1) relatively prime
    Assert SeeIfTheyArePrime(1 a) relatively prime
    Assert SeeIfTheyArePrime(-1, a) relatively prime

For 1,000 tries pick random a that isn't 1 or -1 and then:
    Assert SeeIfTheyArePrime(a, 0) relatively prime
    Assert SeeIfTheyArePrime(0, a) relatively prime

For 1,000 tries pick random a and then:
    Assert SeeIfTheyArePrime(a, -1,000,000) =
        IsRelativelyPrime(a, -1,000,000)
    Assert SeeIfTheyArePrime(a, 1,000,000) =
        IsRelativelyPrime(a, 1,000,000)
    Assert SeeIfTheyArePrime(-1,000,000, a) =
        IsRelativelyPrime(1,000,000, a)
    Assert SeeIfTheyArePrime(-1,000,000, -1,000,000) =
        IsRelativelyPrime(-1,000,000, -1,000,000)
    Assert SeeIfTheyArePrime(1,000,000, 1,000,000) =
        IsRelativelyPrime(1,000,000, 1,000,000)
    Assert SeeIfTheyArePrime(-1,000,000, 1,000,000) =
        IsRelativelyPrime(-1,000,000, 1,000,000)
    Assert SeeIfTheyArePrime(1,000,000, -1,000,000) =
        IsRelativelyPrime(1,000,000, -1,000,000)

7.) Stephens 8.3
I used black-box testing because there was no statement about how the IsRelativelyPrime method works. We could use either a white-box or gray-box technique for testing if we had been told how the IsRelativelyPrime method works. An exhaustive test would be unfit because we have such a big range of values from -1,000,000 to 1,000,000.

8.) Stephens 8.5
I needed to implement restrictions for the values of a and b when I wrote the IsRelativelyPrime method, because the method wasn't equipped to handle the max and min possible integer values. Testing pushed me to think about this restriction, as testing often does with reminding programmers about the special cases that arise that we often forget initially.

9.) Stephens 8.9:
Exhaustive tests are black-box tests because they don't depend on knowing what is happening inside the method that they are testing.

10.) Stephens 8.11
Using the testers in pairs you can calculate the three Lincoln indexes:

-Bob and Carmen: 5 x 4 / 2 = 10
-Alice and Bob: 5 x 5 / 2 = 12.5
-Carmen and Alice: 4 x 5 / 1 = 20

If you take the average of the three calculations above you get ~14. Therefore, there are about 14 bugs. You should also continue to count each bug found so you can revise your estimate as more information comes in.

11.) Stephens 8.12
If the two testers don't find any bugs in common than the Lincoln estimate's equation would call for us to divide by zero which gives us an inadequate result that provides no information about how many bugs there are. If you wanted to get a lower bound estimate for the number of bugs you could pretend the testers had found 1 bug in common and follow the Lincoln estimate equation.